Semantic Locality and Context-based Prefetching Using Reinforcement Learning

Leeor Peled[†] Shie Mannor[†] Uri Weiser[†] Yoav Etsion[†§]
[†]Electrical Engineering [§]Computer Science
Technion — Israel Institute of Technology
{leeor@tx, uri.weiser@ee, shie@ee, yetsion@tce}.technion.ac.il

Abstract

Most modern memory prefetchers rely on spatio-temporal locality to predict the memory addresses likely to be accessed by a program in the near future. Emerging workloads, however, make increasing use of irregular data structures, and thus exhibit a lower degree of spatial locality. This makes them less amenable to spatio-temporal prefetchers.

In this paper, we introduce the concept of Semantic Locality, which uses inherent program semantics to characterize access relations. We show how, in principle, semantic locality can capture the relationship between data elements in a manner agnostic to the actual data layout, and we argue that semantic locality transcends spatio-temporal concerns.

We further introduce the context-based memory prefetcher, which approximates semantic locality using reinforcement learning. The prefetcher identifies access patterns by applying reinforcement learning methods over machine and code attributes, that provide hints on memory access semantics.

We test our prefetcher on a variety of benchmarks that employ both regular and irregular patterns. For the SPEC 2006 suite, it delivers speedups as high as 2.8× (20% on average) over a baseline with no prefetching, and outperforms leading spatio-temporal prefetchers. Finally, we show that the context-based prefetcher makes it possible for naive, pointer-based implementations of irregular algorithms to achieve performance comparable to that of spatially optimized code.

1. Introduction

Memory prefetchers identify regularities in memory access streams in order to fetch data to the caches before a demand access is issued. Most modern prefetchers identify such regularities by leveraging the spatio-temporal memory locality exhibited by common workloads.

ISCA '15, June 13 - 17, 2015, Portland, OR, USA

ACM 978-1-4503-3402-0/15/06 ...$15.00
http://dx.doi.org/10.1145/2749469.2749473

Irregular data structures and algorithms, on the other hand, exhibit lower degrees of spatial locality and are thus less amenable to spatio-temporal prefetchers. As a result, programmers are burdened by the need to map irregular structures and algorithms onto spatial memory layouts (e.g., adjacency matrices representing graphs, array-based binary trees) to improve application performance.

In this paper we introduce the concept of *semantic locality*, a high-level abstraction of data locality that represents relations between data objects as reflected by the semantics of the data structure or the traversal algorithm. For example, elements in a linked list exhibit semantic locality, regardless of whether the list is implemented as a linked data structure or mapped to an array. We argue that this view of locality is inherent to program semantics and transcends spatial implications.

We further propose the context-based prefetcher, which captures machine and program state and uses *reinforcement learning* to approximate semantic locality and predict future memory accesses of irregular data structures and algorithms. The context-based prefetcher tracks the machine context by capturing a number of software attributes, obtained through compiler-injected hints, and using them together with some hardware attributes, captured by the CPU. Examples of compiler injected hints include the type of memory indirection, address offsets, and object data types, whereas hardware attributes include the program counter, branch history, and register contents. The hints and attributes are used to train and execute an algorithm that is based on the contextual bandits model to identify high-level relations between context states and memory addresses and to predict future memory accesses.

We implement the context-based prefetcher and evaluate it using the gem5 simulator [5], running an out-of-order x86 processor model. We tested the prefetcher on a wide range of benchmarks that employ both regular and irregular patterns, including the SPEC2006 [28] suite, benchmarks from the Graph500 [19], PBBS [25] and HPCS [3] suites, as well as a number of μbenchmarks. A modified LLVM compiler was used to identify and inject semantic hints.

Compared to a baseline with no prefetching, our prefetcher is shown to deliver speedups as high as 4.3× over our full set of benchmarks (32% on average), and up to 2.8× over the SPEC2006 suite alone (20% on average). Its average speedup over the entire set is 76% better than that delivered by leading spatio-temporal prefetchers. The context-based prefetcher also makes it possible for naive, unoptimized irregular codes to

deliver performance comparable to that of implementations hand-tuned for spatial locality and run with modern prefetchers.

Our contributions in this paper are as follows:

- We introduce the concept of *semantic locality*, a high-level abstraction of data locality that is based on inherent program semantics rather than memory layout.
- We present the context-based prefetcher, which approximates semantic locality by using machine context (hardware and software) as features for reinforcement learning.
- We propose an implementation of the context-based prefetcher and evaluate it using the gem5 simulator.

The remainder of this paper is organized as follows. Section 2 describes semantic locality and how it may be derived from contextual information, Section 3 discusses related work. Section 4 describes the proposed context-based prefetcher, and Section 5 discusses its implementation. Section 6 describes the simulation methodology, Section 7 describes our evaluation, and we conclude in Section 8.

2. Semantic Locality & Context-based Prefetch

The spatial locality phenomenon characterizes programs due to the linear nature of memory systems. Many common data structures and algorithms exhibit linear behavior. In addition, many structures and algorithms have been explicitly mapped to sequential memory layouts for performance optimization, in order to benefit from the linear structure of memory systems and the advantages it provides (e.g., block and page alignment, prefetching).

However, because spatial locality is an artifact of a physical machine attribute, we ask whether a more generalized form of locality can be identified and used, and what additional information it requires. In this section, we propose the concept of *semantic locality*, in which locality is viewed as a characteristic of program semantics rather than an artifact of machine implementation.

2.1. Locality as an artifact of program semantics

The key concept of *semantic locality* is that memory elements should be associated by the access order dictated by program semantics, namely by the program's data structures and traversal methods. This view logically associates data elements regardless of the physical data layout. We say that two data elements are *semantically adjacent* if the process of accessing one element by the program will likely lead to accessing the second soon after (the relation is directional). In tree or graph structures, for example, semantic locality connects adjacent edges or vertices along any potential path through the graph.

For example, Figure 1 demonstrates the distribution of memory accesses over time for a naive implementation of the insertion sort algorithm using a linked list. The figure shows the data accesses indexed by both real memory addresses (top), and the logical list indices (bottom). As new items are dynamically allocated and inserted at random points, the list quickly

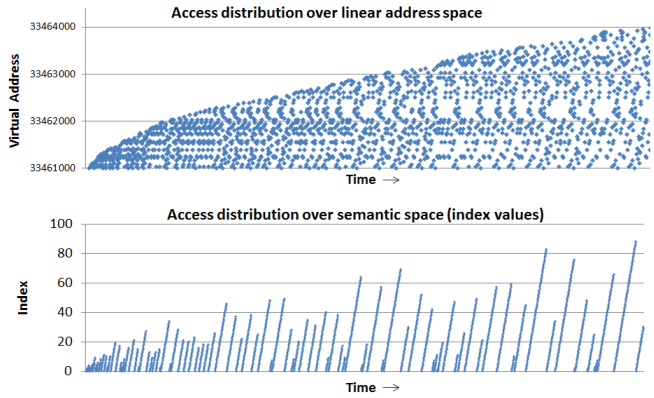

Figure 1: Memory accesses for list insertion sort (100 random elements). Accesses are mapped using real memory addresses (top) and logical list indices (bottom). Even though the list elements are created at random with no spatial locality, the logical traversal is always semantically linear.

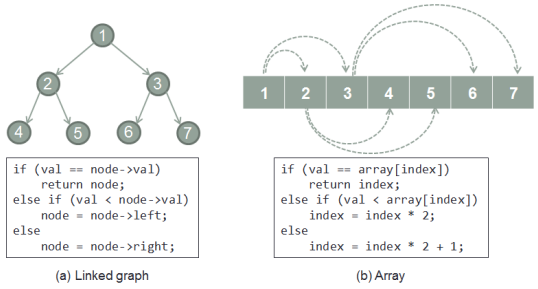

Figure 2: Binary search over a sorted binary tree.

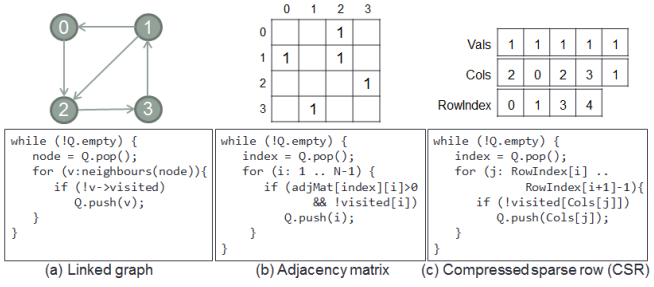

Figure 3: Breadth-first search (BFS) traversal over a graph.

loses its consecutive order in memory (top), rendering spatial prefetchers mostly ineffective. On the other hand, the traversal does retain a perfectly recurring logical pattern, as sorted elements are traversed over and over in the same order on every insertion (bottom). This pattern is obvious only when viewed in the abstract semantic representation.

Figures 2 and 3 illustrate that data layout is not an inherent algorithmic property but rather an artifact of the implementation. The figures present multiple possible implementations for two well-known algorithms — a binary search tree and a breadth-first graph traversal — and the resulting data layouts. Even though the code segments shown provide the same functionality, the degree of spatio-temporal locality will vary

Contextual hint	Description	Source
Instruction pointer (IP)	The most common context element used in IP-based prefetching. The IP identifies a specific load site in the code that would be used for related load actions (same structure, same purpose).	Hardware
History of recent memory accesses	Previous addresses may indicate future ones in recurring patterns, but this feature comes with a risk of overly localized learning and must be used sparingly.	Hardware
Branch history	Hints as to the current control flow, which may, in some cases, indicate a specific path along a diverging data structure.	Hardware
Data stored in general registers	If the traversal decisions depend on some external value stored outside the data structure (e.g., a value being searched), this could provide a useful breakdown between traversal paths.	Hardware
Previously loaded data	Data acquired in previous accesses (globally or within the same partial context)	Hardware
Type information	Unique enumeration of object types, allowing a distinction between data elements in compound structures (e.g. edges vs. vertices in a graph)	Compiler
Link offsets	Offsets within objects, internal layout for pointers or indices used to identify adjacent elements in the data structure	Compiler
Form of reference	Pointer dereference operator (“.”, “->” or “*”), array index, etc.	Compiler

Table 1: Workload attributes/features that compose the context of a memory access.

depending on the implementation.

The graph in Figure 3 illustrates the concept of *semantic adjacency*. We can see that node #2 semantically follows node #1, but node #3 does not immediately follow node #0. The same semantic relations may also be derived from the other implementations of the data structure. Naturally, semantic adjacency can be extended to indirect relations, in a manner similar to the weakening correspondence of second order adjacencies in spatio-temporal locality (which affect prefetcher effectiveness and caching efficiency in current systems).

Semantic locality is, therefore, a characteristic of the memory access semantics, where accesses are considered adjacent if they are related through a given sequence of actions, which the program performs in order to get from one data element to another. This sequence may include operations over any data values available within the code, memory data fetched during the previous accesses, or any combination thereof. These operations, while differing between implementations of the same task, represent the layout-specific part of the access stream pattern. What remains is a logical representation of the data structure and how it is traversed by the algorithm.

Semantic locality describes the data connectivity at a level that transcends implementation details. This distinction is critical for prefetcher design, since the abstraction layers used to implement an algorithm or a data structure add code that can be viewed as metadata. As a result, the memory stream that is observed by the execution core, and in which prefetchers identify access patterns, is an obfuscated, reordered mix of real and metadata accesses. This is also where semantic adjacency differs from *temporal correlation* [6, 8], which associates addresses that are accessed in temporal proximity. Semantic locality, in contrast, directly roots data correlations in semantic states of the program and not on the memory access stream generated by a specific implementation.

It should be noted that semantic locality does not require true data dependency. In non-linked data structures, for example, such as simple arrays and matrices, a sequential access may depend on a running index and not on the outcome of recent accesses (and can also be prefetched by simple stride prefetchers). However, we still preserve weak data dependency through the index variable that is being incremented, and consider such cases as manifestations of semantic locality. Semantic locality can, therefore, be viewed as a generalization of spatial locality, since adjacent addresses can be obtained using simple arithmetic.

2.2. Layout-agnostic representation

Figures 2 and 3, as discussed above, illustrate how common algorithms can be implemented using different data layouts. For example, a binary tree can be implemented as a linked graph or mapped onto an array representation. Typically, the second option is preferred due to its better spatial locality, even though it requires obfuscated index calculations. Graphs share the same problem. While they are easier to implement and manipulate as linked data structures, virtually all applications and benchmarks tailored for high performance (e.g., Graph500) use spatial alternatives such as adjacency matrices or compressed sparse row/column formats (CSR / CSC).

Semantic prefetching can allow programmers to use naive (linked) implementations by eliminating spatial considerations from the implementation process. Concretely, if hardware can capture the semantic relations between the elements in a data structure, regardless of their memory layout, the performance penalty of choosing the simple and straightforward linked implementation will be greatly alleviated.

2.3. Approximating semantics with runtime contexts

Our basic premise in this paper is that memory access streams exhibit semantic traversal patterns. However, these streams comprise a mixture of substreams, some representing true semantic traversals over structured data, and some generated by the implementation of the traversal (including secondary effects such as stack accesses, register spilling or paging). To identify the semantic relations discussed above, it is necessary to isolate the memory access substreams that reflect the paths along data structures. Once these chains of accesses are identified, we can track their progress, so that we may use it when another similar traversal occurs, with similar path choices.

Encoding semantic locality, however, is not trivial for the programmer, let alone for the compiler and other automatic tools. Even though hardware can track memory accesses at runtime, it does not have the information to interpret the semantics of each address. Moreover, identifying program semantics is particularly difficult when relying only on the executable binary, which lacks type and symbol information, and whose code is mangled by compiler optimizations. Consequently, we can only approximate semantic locality.

We choose to approximate program semantics by applying a reinforcement learning algorithm to the sequence of program and machine states that exist when a memory access is issued. The reinforcement learning algorithm identifies correlations between the states and future fetched addresses.

To this end, we have identified a set of machine and workload attributes that can serve as classification features for the learning algorithm. A small set of program and machine state attributes will compose the *context* of a memory access, enabling the learning mechanism to identify semantic streams. The set of context attributes is summarized in Table 1.

The set of attributes used to construct the context of a specific memory access is selected dynamically from the full list of attributes. Dynamic selection is required since using too many attributes would risk overfitting and break down unique streams into incoherent components. Since different streams would benefit from different context attributes, these must be selected for each memory access separately. Given the complexity of determining the correct attributes, the dynamic process must also be adaptive and depend on feedback from the runtime environment. This process is further explained in Section 4, which describes the context-based memory prefetcher.

3. Related work

The best known artifact of semantic locality is spatio-temporal locality, which is targeted by common prefetchers.

Nesbit and Smith [20] proposed the *global history buffer* (GHB) prefetcher, which correlates consecutive elements within the recent history of accesses, assuming they can be used later. The GHB framework has multiple flavors, correlating addresses and deltas, and indexing the prefetcher based on the accessed address (Gloabl) or the localized program counter (PC). Jain and Lin [11] enhanced the prefetcher by focusing on correlated addresses (seen consecutively at runtime) and adding basic context information (PC).

Somogyi et al.[27] proposed the *spatial memory streaming* (SMS) prefetcher, which targets spatial relations. SMS correlates spatial patterns that are not necessarily consecutive. The prefetcher identifies such patterns using contextual data (PC and region offset), and was later extended to use temporal cues as well [26]. These studies show that contextual attributes can effectively identify non-trivial spatio-temporal access patterns.

Semantic locality is not limited to the spatio-temporal domain and extends to linked data structures. Roth et al. [23], Bekerman et al. [4], and later Roth and Sohi [24] explicitly targeted linked data structures. These prefetchers track recurring memory access and generate jump pointers into irregular data structures. Bekerman et al. also advocated using context information for address prediction, albeit in a limited scope.

The use of explicit compiler information is critical in uncovering semantic locality. Wang et al. [30] and Al-Sukhni et al. [1] used static analysis to inject prefetch instructions. While effective, these prefetchers rely only on static analysis and do not incorporate dynamic runtime information.

Several other studies have employed machine learning to enhance microarchitectural components. Joseph and Grunwald modeled the memory access stream as a Markov process [14], using only the address as the process's state. The model does not use other context information, which greatly limits its scalability to predict diverging paths on the Markov graph.

Outside the cache subsystem, machine learning was used by Ipek et al. [10] and by Mukundan et al. [18] to enhance the scheduling of requests in the memory controller. The study defined a finite set of memory controller states and potential actions and used reinforcement learning to train the memory controller. In another domain, Jiménez and Lin [13] introduced the Perceptron branch predictor, which uses the Perceptron machine learning algorithm.

In summary, common prefetchers focus on spatio-temporal locality to identify memory patterns, while others explicitly target linked data structures. In contrast, we argue that semantic locality is inherent to algorithms and data structures, and that spatio-temporal locality and linked data layout are artifacts of this higher-level concept. We propose to use reinforcement learning methods to identify patterns in contextual features provided by both the compiler and the hardware.

4. Context-based Prefetching Using Reinforcement Learning

Reinforcement learning, in its most general form, considers the case when the decisions made by an agent affect the future state of the system or the agent. The decisions are typically driven by a positive or negative reinforcement that is obtained by the agent in form of reward and may be delayed by the natural dynamics of the system. A central challenge in reinforcement learning is the so called temporal credit assignment problem: to which actions in the past should the agent attribute positive or negative reward? Another challenge in reinforcement learning is how to estimate the effect (immediate and long terms) of actions that are not chosen? The agent only witnesses the actions that were actually taken and can only estimate what could have had happened if other actions would have been taken. A third challenge concerns finding the most informative representation of the state dynamics, that would lead to the best prediction of the reward of a given policy. These three challenges distinguish reinforcement learning from other sub-fields of machine learning. In this paper we employ a specific reinforcement learning model called *contextual*

bandits [2, 17]. In this model, the agent observes a context that is determined exogenously and then makes a decision. We focus on estimating the temporal credit assignment and on handling the issue of delayed reward.

4.1. Prefetching and Contextual Bandits

The proposed prefetcher determines what *action* to take for a given system *context*. The system *context* comprises the set of hardware and software attributes listed in Table 1, while each *action* to be taken is the prefetch of a single memory address (or avoiding a prefetch altogether). The problem-specific definitions of *context* and *action* determine the magnitude of the state and action spaces. Our mapping of the prefetcher to a machine learning problem yields very large state and action spaces: the state space comprises the cross-product of all possible feature values (i.e., all rows in Table 1), and the action space consists of all virtual addresses. Furthermore, some states may evolve during runtime, possibly changing the optimal action.

The prefetcher searches for beneficial memory addresses by examining arbitrary context-address associations that do not necessarily manifest a true semantic relation. The process is thus unsupervised, since the training only relies on eventual *feedback* that either strengthens or weakens a *context-address* association. Our online *feedback* mechanism is based on the observation that the usefulness of prefetching an address can be determined by tracking whether it was accessed within a predefined window of memory accesses (e.g., the 128 memory accesses following the prediction). Accurate predictions get feedback in the form of positive rewards, and false predictions get feedback in the form of negative rewards. The size of the reward depends on the timeliness of the prediction. The feedback is processed in a lazy manner (but within a bounded timeframe), while prefetch decisions are made in real-time based on existing knowledge.

Contextual bandits [2, 17] is the simplest reinforcement learning model that matches these restrictions and is used extensively to study exploration/exploitation tradeoffs. This model is a generalization of *multi-armed bandits* [15], a classical machine learning model. The multi-armed bandits model is a feedback-driven method that attempts to find a policy that guarantees the best reward over time for a large set of possible actions, where it is assumed that a selected action does not affect future states. The *contextual bandits* model extends the multi-armed bandits model to use local, dynamic state information for each decision rather than a single global decision policy.

Importantly, both models allow either selecting an action known to be beneficial, or selecting an action whose benefit is unknown, but which might provide a bigger reward and thus extend the space of known actions (e.g., periodically select a random action). Our specific exloration technique is based on the common ε-greedy approach [31] (choosing a random address from the set of previously correlated ones at probability ε on each step). We add some dynamic adaptation based on prediction accuracy, thereby reducing the level of exploration as the predictor begins to converge, similar to the proposal by Tokic [29].

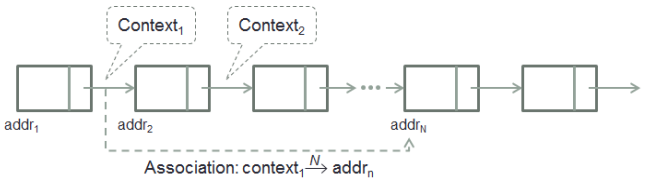

Figure 4: Associating a context with an address for a linked list. Inserting elements between C and A may slightly change their distance but will not break the semantic relation.

We apply a similar approach by generating shadow prefetch operations in parallel to real ones. Shadow operations are not dispatched to the memory system and are only tracked by the prefetcher. These operations allow the prefetcher to gather feedback about the benefit of alternative actions (memory addresses) for a given state (context), without incurring a penalty for low confidence prefetches.

4.2. Context-based prefetching

The context-based prefetcher approximates semantic adjacencies (and thereby semantic locality). The prefetcher maintains a history of observed contexts (the prefetch window), and associates each context with a set of addresses that will likely be accessed some time after the context is encountered.

Formally, $C \xrightarrow{N} A$ denotes that address A is observed N accesses after the context state C was observed. The relation indicates that whenever context C is encountered, there is some likelihood that address A will be accessed N memory accesses in the future. Figure 4 depicts such a relation for a linked list.

The prefetcher closes a feedback loop by applying a reward (either positive or negative) to every memory access. The reward either strengthens or weakens the association $C \xrightarrow{N} A$ that binds the currently observed address to a previously observed context. The reward function is discussed in Section 4.3.

Algorithm 1 depicts a simplified version of the training algorithm, which is executed on every memory access. The algorithm comprises three operations that execute in parallel:

Data collection: Expand the set of addresses associated with each recently observed context. The prefetcher iterates over the context history queue and captures the contexts whose depth is within the (predefined) prefetch window. This associates the current address with the previously observed contexts. In practice, we only sample a subset of the context-address pairs in order to reduce the overhead of the operation.

Prediction: Look up the current context. If a similar context was already encountered, and it has a list of future addresses associated with it, the prefetcher picks the highest scoring one and adds it to the prefetch queue. The prefetcher may add more than one address if there are several high scoring

Algorithm 1 Training function (*addr, context*)

```
1: key ← hash(context)
2: for a ∈ (previous_accesses) do
3:     prev_key = hash(a.context)
4:     states[prev_key].add_ptr(addr)
5: end for
6: if key ∈ states then
7:     best_addr ← max_score{states[key].ptrs}
8:     prefetchQ.push(best_addr, key)
9: end if
10: for p ∈ (prefetchQ) do
11:     if p.addr = addr then
12:         reward ← f_Reward(p.index)
13:         states[p.key].ptrs[addr].score += reward
14:     end if
15: end for
16: next_pref ← prefetchQ.pop()
17: send_prefetch(next_pref)
```

candidates. It may also generate shadow prefetch operations for training. If a selected address already exists in the prefetch queue due to an earlier prefetch, the prefetcher will add it to the prefetch queue again as a shadow prefetch to train another context-address pair. Addresses in the prefetch queue that are not marked as shadow prefetches are sent to memory.

The number of addresses that can be pushed into the prefetch queue is dynamically throttled. The number of addresses that get prefetched for a given context is determined by the accuracy of the prefetcher (average hit rate in the prefetch queue). Furthermore, prefetch operations may be skipped if the memory system is stressed (e.g., if the number of available MSHRs falls below a predefined threshold), thereby converting them to shadow operations.

Feedback: The prefetcher scans the prefetch queue and identifies the contexts that have predicted the current address. The selected contexts' scores are then updated using a reward function that accounts for the prediction accuracy, namely the distance at which the address was actually used (measured as the depth in the prefetch queue). The prefetcher also updates the outcome of shadow operations.

4.3. The reward function

The learning mechanism must only accept semantic relations whose distance is within the effective prefetch window. Prefetching data that will be accessed in the very near future is useless, since the operation will not complete before the demand fetch is issued. Similarly, prefetching data that will be accessed in the distant future is also useless, since the data will likely be evicted from the caches prior to the demand access.

In order to correctly identify that a context C is semantically correlated with an address A within the effective prefetch window, the prefetcher rewards repetitions with similar prefetch distances. Specifically, a recurring relation will be strengthened by promoting address A over other potential addresses associated with context C. But even though a context C and the related address A may have a strong semantic relation, their distance may vary due to different control flows. In addition, instruction reordering in an out-of-order pipeline may also affect the prefetch distance. The reward function must thus be flexible to accommodate such variations.

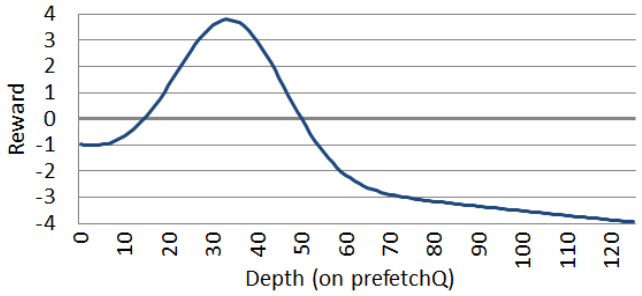

Figure 5: Reward function for context-based prefetcher.

Our reward function, depicted in Figure 5, is bell shaped to accommodate the effective prefetch window. The bell shape provides a graceful degradation of scores over the prefetch window to align predictions inside the window. The function's negative edges enable the prefetcher to demote context-address pairs that were once associated inside the effective prefetch window, but whose relation has shifted outside the window (suggesting they are no longer semantically adjacent, or that the initial pairing is not a recurring one).

The prefetch window is aligned to the target prefetch distance, which is measured in memory accesses. Since data is prefetched to the L1 cache, the distance is derived from the average L1 miss penalty. In our 2-level cache system (whose parameters are listed in Table 2), that penalty (in cycles) is:

$$L1\ miss\ penalty = L2\ latency + L2\ miss\ rate \times DRAM\ latency.$$

The average penalty is then converted to a number of memory accesses; this number is based on workload parameters such as *IPC* and the probability of encountering a memory operation:

$$prefetch\ distance = L1\ miss\ penalty \times IPC \times Prob(mem.\ op.).$$

Naturally, the target prefetch distance varies for different workloads. For the benchmarks evaluated in our study, we found that the target distance ranges between ∼10–90 memory accesses, with an average of ∼30 accesses. The bell shaped reward function is thus centered around the average workload, and its bell-shape enables a single function to accommodate diverse workloads with varying degrees of success.

4.4. Context hashing and reduction

The attributes/features tracked by the prefetcher (Table 1) are concatenated into a bit array and hashed. The resulting hash value represents the current context.

But using the full set of attributes might sometimes cause overfitting, and it may be beneficial to only use a subset of the attributes. Conversely, the set of active attributes is sometimes insufficient and should be extended to break down a single context into multiple, distinct contexts.

We therefore propose a 2-level indexing method to dynamically control the active set of attributes. Specifically, the full hash value is used to index a *Reducer* table, which holds a bitmap of the active attributes that should be used to index the context states table. The active attributes are then re-hashed to generate the index for the context states table.

The prefetcher dynamically controls the set of active attributes for each context by means of an *overload* score, which identifies when a reduced context has too many prefetch candidates. This scenario suggests that many full contexts map to a single reduced context because they differ only in inactive attributes. The score is calculated on every context lookup.

When an overload is encountered, the prefetcher activates the first inactive attribute in the list of attributes and thereby splits the reduced context into multiple ones, which are distinguished by different values in the newly activated attribute. Similarly, when the context table entries are underloaded, indicating that contexts are distributed over too many unique states, the prefetcher can disable active attributes to merge context states.

5. Prefetcher Architecture

The main structures and the flow of the proposed prefetcher are depicted in Figure 6. The architecture is designed to follow the flow described in Section 4 and outlined in Algorithm 1, and it thus consists of three components that operate in parallel:

1. The *collection unit* tracks context history and, using the current memory address, creates context-address pairs.
2. The *prediction unit* looks up the current context in the context table to generate prefetches.
3. The *feedback unit* closes the reinforcement learning loop by updating the scores of previously encountered contexts, if the current memory access hits any prefetches they issued.

Collection Unit The main structures in the collection unit are two direct-mapped tables:

- The *context-states table* (CST) binds contexts (or hashed and compacted representations thereof) to addresses associated with them, and stores the scores for ranking these associations according to the provided feedback. The CST entries provide the space of possible *actions* for the exploration/exploitation of each stored context. Each CST entry also tracks the number of reducer entries that point to it through the indexing process.
- The *Reducer* performs online feature-selection by choosing the system attributes to be included in the system context. As explained in section 4.4, this is determined according to the reduction ratio, by deactivating attributes when a context in the CST is pointed to by too few reducer entries (implying overfitting), or activating attributes when the reduction is too aggressive and too many entries point to the same reduced context (context overload). Each reducer entry holds a bit for each attribute. Once the active attributes have been determined, the reducer will rehash the partial context into a compact representation to obtain the final CST index.

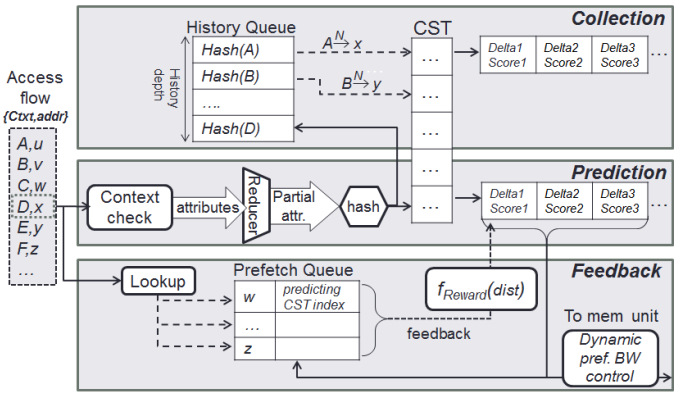

Figure 6: The layout of the context-based prefetcher and the information flow within.

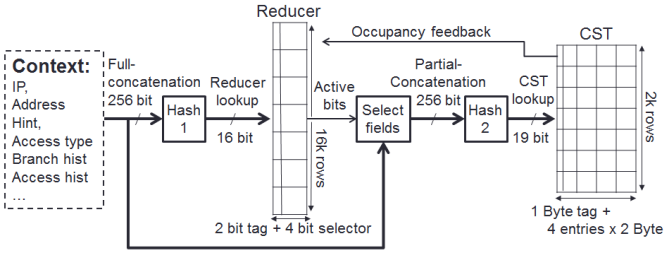

Figure 7: Reducer and CST indexing scheme.

Figure 7 describes the mapping of a context state into a CST entry. As described in the previous section, the attribute values are concatenated into a bit array and are hashed to produce a 16 bit value. The value's lower 14 bits are used to access the reducer and fetch the active bits for this entry, while the remaining bits are used as a tag. The active bits are then used to generate the partial context, whose hash value is used to address the CST. The second hash value consists of 19 bits, 8 of which serve as a tag. Notably, this structure may yield conflicts, mainly in the reducer, but we have observed that such conflicts have little impact on the prefetcher's performance.

In order to reduce the storage space, the CST stores address deltas (relative to the address of the relevant context). Each CST entry may hold 4 context-address associations, each consisting of a 1-byte delta of cache line granularity (able to point within a range of up to 8kB in each direction), and a 1-byte integer score. The CST employs a score-based replacement policy, which benefits pairs that gained positive rewards.

The current context is pushed to the *History Queue*, which stores the sequence of observed contexts that are waiting to be associated with impending memory addresses. In parallel, the current address is associated with older contexts in that queue. The generated context-address associations are added to the CST in order to expand the exploration space, potentially evicting previously stored associations that have accumulated the lowest scores.

To fully explore the learning space, we should associate the currently accessed address with all contexts in the history

queue. However, to avoid using a fully-associative structure, we use a probabilistic lookup to only search a small number of entries. The collection unit samples the queue at a number of predefined depths. This form of probabilistic lookup was shown to be effective in mitigating overheads in other memory system components due to the skewed distribution of memory accesses [7, 21, 22].

Prediction Unit The prediction unit receives the current hashed context and looks it up in the CST to produce a list of potential prefetch addresses. The unit then prefetches the addresses with the highest scores. The exact number of prefetch operations that are issued is determined dynamically based on the prefetcher's hit rate and the load on the memory system. To facilitate learning even in infrequent scenarios (in accordance with the *contextual bandits* model), the unit occasionally chooses a random address from the list and issues a shadow prefetch. The current context and the prefetched addresses are then pushed into the feedback unit, where they will be stored until feedback is collected. Finally, the prefetched addresses are returned to the memory unit for dispatching.

Feedback Unit The main structure in the feedback unit is the *Prefetch Queue*, which is used to reward recently predicted context-address associations. The queue holds the most recent predictions and, on a memory access, it is searched to identify all the contexts that triggered a prefetch to the currently accessed address. The depth at which the address was found in the queue is used by the reward function (Figure 5) to update the score of the context-address pair in the CST. In order to track contexts that predicted the access too early and provide negative feedback, the prefetch queue has to be larger than the useful prefetch window. While the effective window's depth stretches up to $\sim$50 accesses in our simulations, we use a prefetch queue with 128 entries. Furthermore, expired entries that are popped from the queue without ever being hit are also used to update the corresponding CST entries with negative scores. Again, we avoid using a fully associative structure by only examining a small number of elements in the prefetch queue on each cycle, and extending the lookup over multiple cycles (queuing the lookups that arrive in the meantime). Our experiments showed that a small queue for pending lookups is enough to sustain peak access rates and, as stated in section 4, the reward delivery may be deferred with no impact on performance.

Notably, the CST may be concurrently accessed by the three prefetcher units. Nevertheless, only prediction requests must be handled in real-time, whereas collection and reward requests can be deferred, as they only update the prefetcher's state for future operations. The CST is, therefore, designed with one read port and one write port, and handling of concurrent requests is serialized to prioritize prediction requests. In addition, the prefetch queue requires a write port and a read port. The write port is used to allocate predictions (handling of concurrent requests can be deferred), while the read port is used to send feedback about matched predictions (during one of the lookup cycles). Similarly, the history queue requires a single write port for pushing new context states, and a read port for reading the predefined entries.

In total, the prefetcher requires a total of 31kB of storage, as described in table 2.

Simulation mode	Sys. emulation, accurate timing, x86
Core type	OoO, 4-wide fetch
Queue sizes	192 ROB, 64 IQ, 256 PRF, 32 LQ/SQ
MSHRs	L1: 4, L2: 20
L1 cache	64kB Data, 32kB Code, 8 ways, 2 cycles access, private
L2 cache	2MB, 16 ways, 20 cycles access, shared
Main memory	2GB, 300 cycles access
Context prefetcher	
CST	2K entries x 4 links (18kB), direct-mapped
Reducer	16K entries, direct-mapped (12kB)
History queue	50 entries x 19 bit context (120B)
Prefetch queue	128 entries of address/context pairs (1.3kB)
Overall size	$\sim$31kB
Competing prefetchers	
GHB (all)	GHB size: 2K, History length: 3 Prefetch degree: 3, Overall size: 32kB
SMS	PHT size: 2K, AGT size: 32, Filter Table: 32 Regions size: 2kB, Overall size: 20kB

Table 2: Simulator parameters.

6. Methodology

We implemented the context-based prefetcher using the gem5 [5] simulator, running an out-of-order x86 model (single core). Table 2 details the system configuration.

We have implemented an LLVM [16] pass to extract semantic code hints. The compiler identifies pointer-based memory accesses to objects (e.g., structs, classes) and enumerates the type of object (each type is assigned a unique value within the compiled program). The compiler also identifies object data members that are pointers. The compiler's backend packs contextual features as 32-bit immediate values that are injected to the generated code using an x86 extended NOP instruction (i.e., NOP with an immediate value). Each memory instruction for which contextual features are available is immediately preceded by a single NOP instruction that carries the contextual hints. When a NOP is decoded in the pipeline, its features are attached to the following memory operation, passed along the pipe, and consumed by the memory unit.

In order to reduce the overhead incurred by the additional NOP instructions, the compiler only injects contextual features for operations that write new values to addresses that are represented as pointers at the program level. This is in contrast to *pointer+offset* accesses (e.g., accessing an object's members through its pointer), which access data that was likely already prefetched by the original access to the base pointer.

We evaluate the context-based prefetcher using a collection of benchmarks that include both regular and irregular memory patterns. The benchmarks were collected from a variety of well-known benchmark suites, including SPEC2006 [28],

Suite	Workloads
SPEC CPU2006 [28]	sjeng, povray, soplex, dealII, h264ref, gobmk hmmer, bzip2, milc, namd, omnetpp, astar libquantum, mcf, sphinx3, lbm
PBBS [25]	suffixArray, BFS, setCover, KNN
Graph500 [19]	Graph500
HPCS [3]	SSCA2(v2.2): CSR / List (array)
μkernels (algorithms)	Prim, listsort, SSCA_LDS (linked version)
μkernels (Data structure traversals)	list, array, hashtest (STL unordered map), maptest (STL RBtree map)

Table 3: Workload and benchmarks used

Graph500 [19], HPCS [3] and PBBS [25]. We also include a set of μbenchmarks that implement several well-known data structures and algorithms with irregular footprints (linked list, binary search tree and Prim's minimum spanning tree algorithm). Table 3 lists the benchmarks used. We used the full SPEC2006 suite, except for benchmarks that include Fortran code and are not supported by *clang*. We also encountered compilation issues with *gcc*, *perlbench* and *xalancbmk*.

We have simulated distinct execution phases for each benchmark (the exact number of phases vary between benchmarks). The execution phases represent programs' execution during their steady state, and they span 50–100M instructions each (experiments validated that the impact of using longer phases is negligible). Simulation phases were selected based on runtime characterization. We used the analysis by Jaleel et al. [12] for the SPEC2006 benchmarks, and a similar, in-house analysis for the other benchmarks.

7. Performance Evaluation

We evaluated the context-based prefetcher by examining its accuracy and convergence and by quantifying its impact on cache and system performance. We also examined how its storage size affected performance. We compared our proposed prefetcher with other high-performance prefetchers: the *global history buffer* (GHB) [20] (the Global Delta-Correlation flavor, or G/DC, as well as the PC Delta-Correlation flavor, or PC/DC) and *spatial memory streaming* (SMS) [27]. A stride prefetcher [9] was also evaluated, but its performance was significantly lower and is thus not shown. The storage size of all prefetchers was scaled to that used by the context-based prefetcher, as shown in Table 2.

7.1. Accuracy and convergence

We used two metrics to evaluate the accuracy of our prefetcher:

- *Learning accuracy*: The accuracy with which the training process converged to the desired prefetch distance, as determined by the depth at which actual demands hit the prefetch queue.
- *Prefetcher accuracy*: the fraction of demand accesses that benefited from the prefetcher (at varying degrees)

We measured the convergence of the learning process by exploring the distribution of hit depths, defined as the number of memory accesses that occur between issuing a prefetch

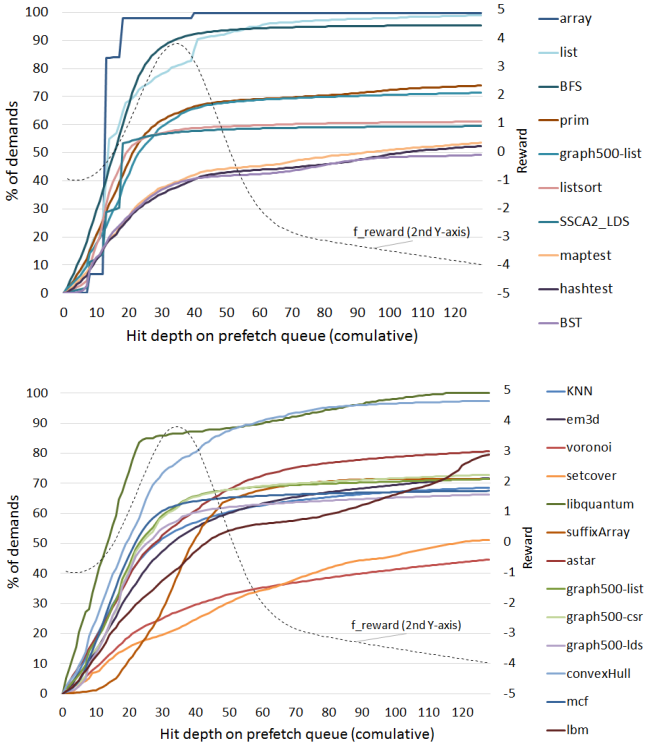

Figure 8: Cumulative distribution of hit depths for the μbenchmarks (top) and regular benchmarks (bottom). The overlayed reward function illustrates the prefetch window.

(or a shadow prefetch) and the demand fetch associated with it. We present a cumulative distribution function (CDF) in which a value of P at depth N means that P% of the predicted lines were hit by a demand access within at most N memory accesses. We expected that the majority of samples will be centered around the range in which the bell in the reward function is positive (Figure 5). This range (18–50 memory accesses) provides the desired distance (for our simulation configuration) by which a prefetch should precede an ensuing demand fetch in order to hide the memory latency.

Figure 8 depicts the CDFs of the hit depths for both μbenchmarks (top), and a subset of the regular benchmarks (bottom). We only show a subset of the benchmarks to reduce clutter. The CDFs manifest both real and shadow prefetches. We focus our discussion on the μbenchmarks, since both CDFs show similar trends.

As expected, a noticeable step in the CDFs is visible, beginning at a depth of 18 accesses, which corresponds to the positive range of the reward function. Nevertheless, the CDF steps are decomposed into a cluster of sub-steps. This decomposition is attributed to fluctuations in the prefetch distance caused by changes in the control flow of recurring code and request reordering in the out-of-order pipeline. The figure also demonstrates prefetch inaccuracies, some of which are caused by the need to continuously train the prefetcher. Up to ~25% of the prefetch operations (~35% for *BFS*) are issued too late and precede the demand fetch by less than 18

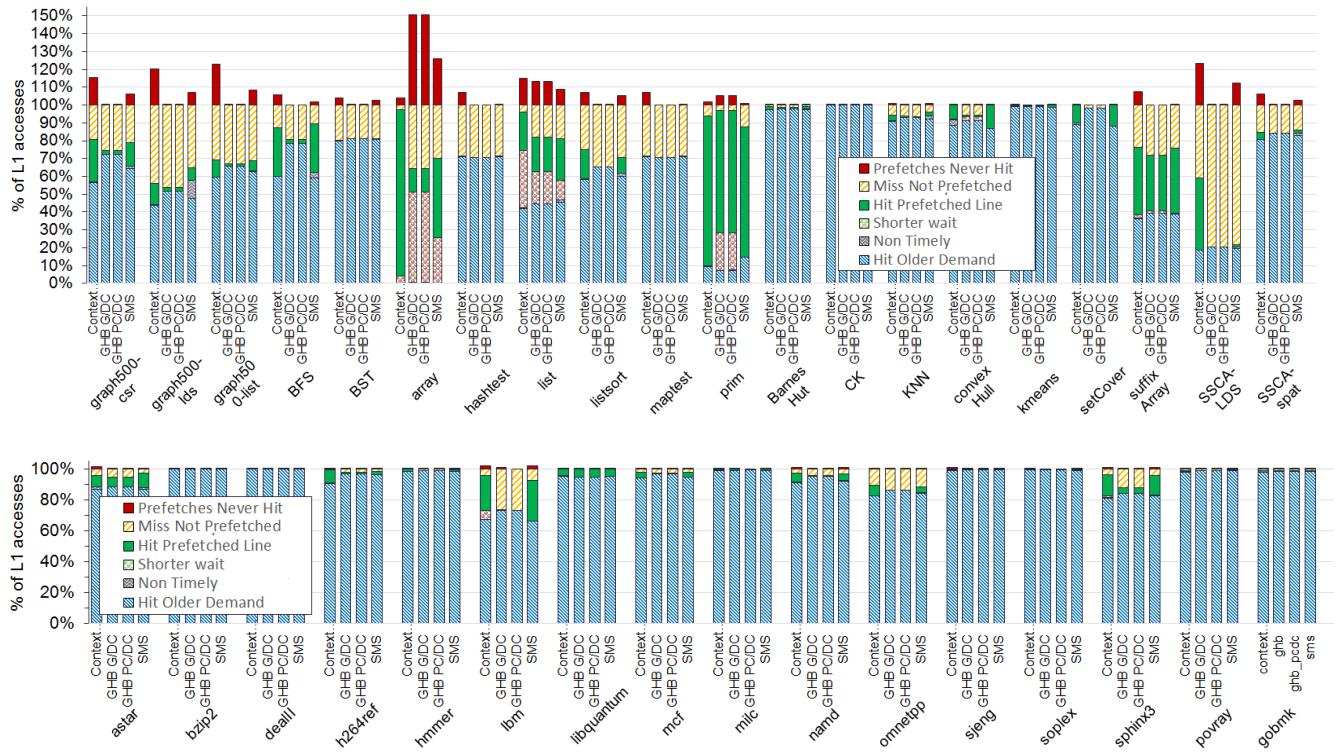

Figure 9: Accuracy and timeliness compared for the evaluated prefetchers.

accesses. When it comes to early prefetches (which precede the demand fetch by more than 50 accesses), we see that the μbenchmarks compose three distinct groups. The top three benchmarks (*array*, *list* and *BFS*) enjoy high accuracy, and less than 10% of their prefetches are issued too early. The next four (*Prim*, *Graph500-list*, *listsort* and *SSCA_LDS*) have a non-negligible fraction of early prefetches, yet these are still bounded in the ~30–40% range. The last three benchmarks represent input dependent lookup operations (*maptest*, *hashtest* and *BST*), which are very difficult to predict, mostly due to their high degree of branching.

The second method for measuring the accuracy of the prefetcher is by classifying the type of benefit provided by the prefetcher for each memory access:

- *Demand hits a prefetched line*: The prefetch was useful, and the demand access hit the cache because of the prefetch.
- *Shorter wait time*: The access missed in the cache but enjoyed a shorter wait time because of an ongoing prefetch.
- *Non-timely*: The prefetcher predicted the address, but a request was not issued to memory before the demand access.
- *Miss not prefetched*: The address of the demand access was not predicted by the prefetcher.
- *Hit older demand*: No prefetch was needed, and the demand access hit in the cache.
- *Prefetch never hit*: The prefetch predicted a wrong address (line was never used while in the cache). These wrong predictions are counted on top of the program's demand accesses, and therefore pass the 100% mark.

Figure 9 categorizes the memory accesses using the different prefetchers. The figure shows that, for the benchmarks that benefit from prefetching, the context-based prefetcher tends to have a larger fraction of the accesses categorized as successful prefetches, either because the data was prefetched on time (*Hit Prefetched Line*) or because the wait time was reduced. This trend is clear in the irregular benchmarks (*Graph500* and *Prim*), in the μbenchmarks, and in the SPEC benchmarks (*h264ref*, *lbm*, *namd*, *omnetpp* and *sphinx3*).

Interestingly, the figure also shows that the context-based prefetcher correctly identifies strict regular patterns (e.g., the *array mu*benchmarks). These results suggest that the prefetcher indeed captures access semantics rather than focusing on a specific access pattern.

7.2. Cache performance

We evaluated cache performance by quantifying the *misses per kilo-instructions* (MPKI) in both L1 and L2 caches. This metric reflects the prefetching benefits for short and medium latency accesses, which correspond to the range targeted by the reward function.

Figures 10 and 11 present the MPKI on each cache level for the different prefetchers. The context-based prefetcher consistently delivers lower MPKI than any other prefetcher. On average, it reduces the L2 MPKI by almost a factor of 4 compared to a system without prefetching, from almost 40 misses to ~10 misses. Finally, it outperforms SMS, the best competing prefetcher, by a factor of 2.

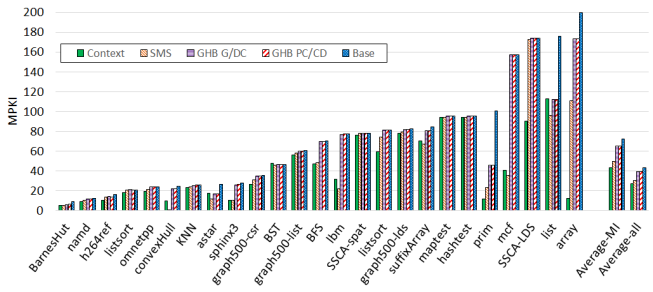

Figure 10: L1 Misses per kilo-instruction (MPKI) for the different prefetchers. We show the most memory-intensive benchmarks (MPKI > 5), as well as the average for all benchmarks.

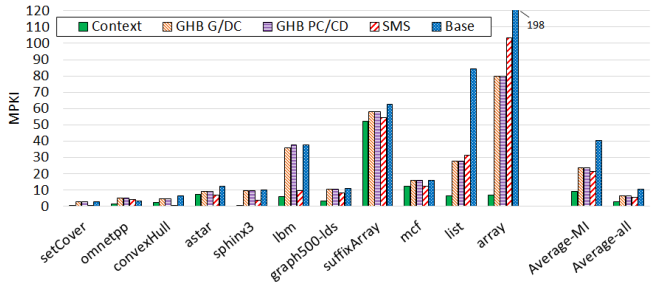

Figure 11: L2 Misses per kilo-instruction (MPKI) for the different prefetchers. We show the most memory-intensive benchmarks (L2 MPKI > 1), as well as the average for all benchmarks.

7.3. Speedup

Figure 12 presents the speedups delivered by the different prefetchers (measured as instructions-per-cycle, or IPC) over a baseline with no prefetching.

The figure demonstrates the effectiveness of the context-based prefetcher, which delivers an average speedup of 32% over a system with no prefetching, and an average speedup of 20% for the SPEC2006 suite alone (bottom of the figure). Furthermore, the prefetcher outperforms the next best performing prefetcher (SMS) by an average of ∼76% (and up to 2.1×), with only a single significant negative outlier (convexHull).

In a few cases where a competing prefetcher outperforms ours (e.g., *convexHull*, *astar*, *suffixArray* and *setCover* benchmarks), we identify three main causes:

- *Pattern depth*: Context-address associations are limited by the depth of the *history queue*. As a result, patterns whose distance is longer than the queue are not identified.
- *Training speed for simple patterns*: While the learning algorithm may capture more elaborate access patterns, the learning takes longer than with a dedicated prefetcher (for example, a stride prefetcher would identify strides much faster than the context-based prefetcher). As a result, the context-based prefetcher cannot ameliorate the training time for short program phases.
- *Fine-grained strides*: Operating at fine address granularity (word size) would introduce too many distinct addresses to the context-based prefetcher and thrash its internal tables. As a result, the prefetcher operates on 32-byte (aligned) blocks and might thus miss patterns that fall inside a block.

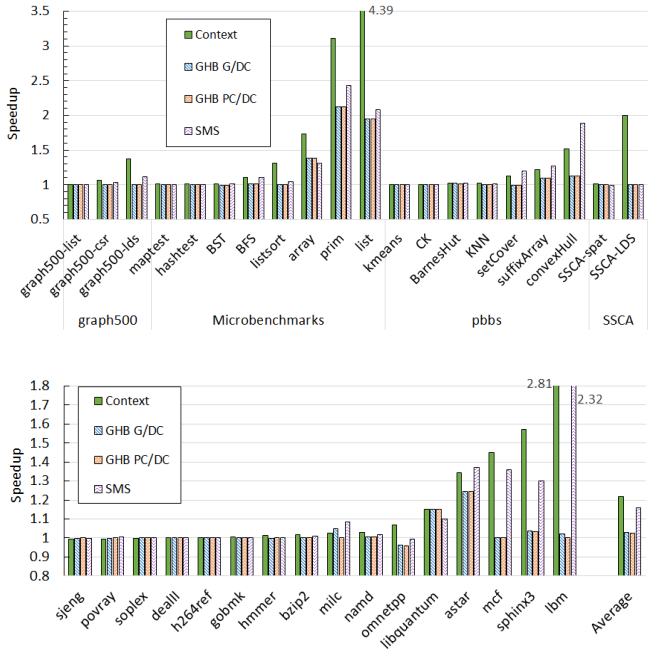

Figure 12: Speedups delivered by the different prefetchers. The baseline is a system without prefetching.

In summary, the experiment demonstrates the benefits of the context-based prefetcher on overall system performance, while exposing a few inherent limitations.

7.4. Prefetcher storage size

There is a tradeoff between the number of contexts the prefetcher can track and the overheads associated with its physical storage size.

Interestingly, while one expects that larger storage sizes would benefit the prefetcher, this is not necessarily the case with a machine learning framework. While maintaining a large number of addresses per context enables tracking multiple potential predictions, it may also limit the odds of selecting an effective prefetch candidate and thus increase the training time. Moreover, even when the set of addresses that can be tracked by the prefetcher is far smaller than the program's entire address space, it may still be sufficient to capture the frequently accessed addresses, or core, of the working set. In fact, the reinforcement learning algorithm increases the odds that the stored elements will be the most useful ones.

Figure 13 shows the effect of storage size on performance. The different configurations scale the number of CST entries, while maintaining the number of reducer entries at 8× that size. The figure shows two lines: the first depicts the effect of the varying storage size on the average performance of the 10 benchmarks that best benefit from the prefetcher (*Top10*), and the second line shows the impact over all benchmarks (*All*).

As expected, increasing the storage size is not necessarily beneficial. The performance benefit of the *Top10* bench-

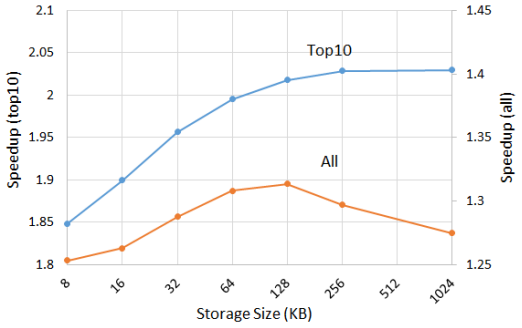

Figure 13: The impact of CST size on the overall speedup.

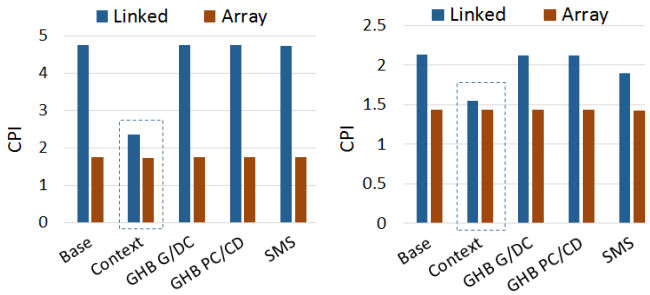

Figure 14: Comparison of prefetcher performance on naive (linked) and spatially optimized, array-based implementations.

mark peaks at a storage size of 256kB (and even decreases at much larger sizes, which are not shown). When examining how the prefetcher storage size affects the performance of *All* benchmarks, we see that the benefit peaks at 64–128kB and then immediately drops. These results show that a storage size of 64kB would probably provide the best overall performance. Nevertheless, since that is considered fairly large for a prefetcher, we used a storage size of ∼31kB in our evaluation and scaled the storage of the other prefetchers accordingly.

7.5. Data-layout agnostic programming

We have shown the benefits of semantic locality and context prefetching over applications that employ non-spatial data structures. However many applications (and most of the common benchmarks) are aggressively optimized for spatial locality, even at the cost of obfuscating the implementation. For example, graphs are often represented as adjacency matrices, and lists and trees are often mapped to arrays.

In this experiment, we explored how the different prefetchers handle both naive and spatially-optimized implementations of irregular data structures.

Figure 14 presents two such case studies. The figure shows the performance (measured as cycles-per-instruction) delivered by the different prefetchers for two algorithms: SSCA2, which finds the betweenness-centrality score of a graph; and Graph500, which mostly executes a breadth-first search (BFS). Both algorithms can be implemented either using a spatial data layout (e.g., arrays) or a linked data layout (e.g., lists, graphs).

The figure illustrates the performance of the different prefetchers for both naive and pointer-based implementations of the algorithms. In both cases, the context-based prefetcher provides the best performance and enables the linked data layouts to enjoy performance comparable to that of spatially optimized code. This is in contrast to all other prefetchers, which distinctively favor spatially-optimized implementations.

In summary, the evaluation demonstrates the performance benefits of the context-based prefetcher, and its effectiveness in approximating semantic locality.

8. Conclusions

In this paper, we introduce the concept of *semantic locality*, which reflects the data correlations inherent to the semantics of algorithms and data structures. Semantic locality provides the underlying reason for predictability in memory access streams. We further argue that other forms of locality, and in particular spatio-temporal locality, are artifacts of mapping semantic locality onto the linear memory system.

Observing that memory access patterns are rooted in the semantics of the program, we propose the *context-based prefetcher*. The prefetcher employs the contextual bandits model of reinforcement learning to approximate program semantics (using a variety of hardware and software attributes) to uncover memory access patterns.

Finally, we evaluate the proposed context-based prefetcher, compare its performance to a number of modern prefetchers and demonstrate its applicability to a variety of benchmarks consisting of both regular and irregular code. We show that the context-based prefetcher can dramatically reduce memory misses (measured as MPKI in the L2 cache) and speed up execution by up to 4.3× compared to a system with no prefetching. We also show that, on average, the prefetcher improves performance by over 32%, exceeding the performance boost delivered by competing prefetchers.

Our work considers a contextual bandit model and ignores the effect of actions not taken. Still, even when considering the simpler contextual multi-armed model we manage to obtain improved performance. We think that finding good representations of the state space (those are usually needed to handle large problems) and policy improvement techniques in the spirit of policy search methods would be instrumental in improving the performance further. We leave this important topic for future research.

Acknowledgment

We thank José Martínez and the anonymous reviewers for their valuable comments and suggestions. This research was funded by the Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI), by the Israel Science Foundation (ISF grant 769/12; equipment grant 1719/12), and by the European Commission (FP7 CIG grant 334258). Y. Etsion was supported by the Center for Computer Engineering at Technion.

References

[1] H. Al-Sukhni, I. Bratt, and D. Connors, "Compiler-directed content-aware prefetching for dynamic data structures," in *Intl. Conf. on Parallel Arch. and Compilation Techniques (PACT)*, Sep 2003.

[2] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire, "The non-stochastic multiarmed bandit problem," *SIAM Journal on Computing*, vol. 32, no. 1, pp. 48–77, 2002.

[3] D. A. Bader and K. Madduri, "Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors," in *Intl. Conf. on High Performance Computing (HiPC)*, Dec 2005. Available: http://dx.doi.org/10.1007/11602569_48

[4] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser, "Correlated load-address predictors," in *Intl. Symp. on Computer Architecture (ISCA)*, May 1999. Available: http://dx.doi.org/10.1145/300979.300984

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 simulator," *Computer Architecture News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. Available: http://doi.acm.org/10.1145/2024716.2024718

[6] S. Chien and N. Immorlica, "Semantic similarity between search engine queries using temporal correlation," in *Intl. Conf. on on World Wide Web*, May 2005. Available: http://doi.acm.org/10.1145/1060745.1060752

[7] Y. Etsion and D. G. Feitelson, "L1 cache filtering through random selection of memory references," in *Intl. Conf. on Parallel Arch. and Compilation Techniques (PACT)*, Sep 2007.

[8] B. Falsafi and T. F. Wenisch, "A primer on hardware prefetching," *Synthesis Lectures on Computer Architecture*, vol. 9, no. 1, 2014.

[9] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," in *Intl. Symp. on Microarchitecture (MICRO)*, Dec 1992. Available: http://dl.acm.org/citation.cfm?id=144953.145006

[10] E. Ipek, O. Mutlu, J. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *Intl. Symp. on Computer Architecture (ISCA)*, Jun 2008.

[11] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *Intl. Symp. on Microarchitecture (MICRO)*, Dec 2013.

[12] A. Jaleel, "Memory characterization of workloads using instrumentation-driven simulation," *Intel Corporation, VSSAD*, 2010. Available: http://www.jaleels.org/ajaleel/workload/

[13] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *Symp. on High-Performance Computer Architecture (HPCA)*, Jan 2001. Available: http://dl.acm.org/citation.cfm?id=580550.876441

[14] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *Intl. Symp. on Computer Architecture (ISCA)*, Jun 1997. Available: http://doi.acm.org/10.1145/264107.264207

[15] M. N. Katehakis and A. F. Veinott Jr, "The multi-armed bandit problem: decomposition and computation," *Mathematics of Operations Research*, vol. 12, no. 2, pp. 262–268, 1987.

[16] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis transformation," in *Intl. Symp. on Code Generation and Optimization (CGO)*, Mar 2004.

[17] L. Li, W. Chu, J. Langford, and R. E. Schapire, "A contextual-bandit approach to personalized news article recommendation," in *Intl. Conf. on on World Wide Web*, Apr 2010. Available: http://doi.acm.org/10.1145/1772690.1772758

[18] J. Mukundan and J. Martínez, "Morse: Multi-objective reconfigurable self-optimizing memory scheduler," in *Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2012.

[19] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, *Introducing the Graph 500*, Cray Users Group (CUG), May 2010.

[20] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in *Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2004. Available: http://dx.doi.org/10.1109/HPCA.2004.10030

[21] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Intl. Symp. on Computer Architecture (ISCA)*, Jun 2007.

[22] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for MLP-aware cache replacement," in *Intl. Symp. on Computer Architecture (ISCA)*, Jun 2006.

[23] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," in *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems (ASPLOS)*, Oct 1998. Available: http://doi.acm.org/10.1145/291069.291034

[24] A. Roth and G. S. Sohi, "Effective jump-pointer prefetching for linked data structures," in *Intl. Symp. on Computer Architecture (ISCA)*, May 1999. Available: http://dx.doi.org/10.1145/300979.300989

[25] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan, "Brief announcement: the problem based benchmark suite," in *Symp. on Parallel Alg. and Arch. (SPAA)*, Jun 2012.

[26] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," in *Intl. Symp. on Computer Architecture (ISCA)*, Jun 2009. Available: http://doi.acm.org/10.1145/1555754.1555766

[27] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *Intl. Symp. on Computer Architecture (ISCA)*, Jun 2006. Available: http://dx.doi.org/10.1109/ISCA.2006.38

[28] Standard Performance Evaluation Corporation, "SPEC2006," http://www.spec.org.

[29] M. Tokic, "Adaptive ε-greedy exploration in reinforcement learning based on value differences," in *KI 2010: Advances in Artificial Intelligence*, ser. Lecture Notes in Computer Science, R. Dillmann, J. Beyerer, U. Hanebeck, and T. Schultz, Eds. Springer Berlin Heidelberg, 2010, vol. 6359, pp. 203–210. Available: http://dx.doi.org/10.1007/978-3-642-16111-7_23

[30] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems, "Guided region prefetching: A cooperative hardware/software approach," in *Intl. Symp. on Computer Architecture (ISCA)*, Jun 2003. Available: http://doi.acm.org/10.1145/859618.859663

[31] C. J. C. H. Watkins, "Learning from delayed rewards." Ph.D. dissertation, King's College, Cambridge, May 1989.